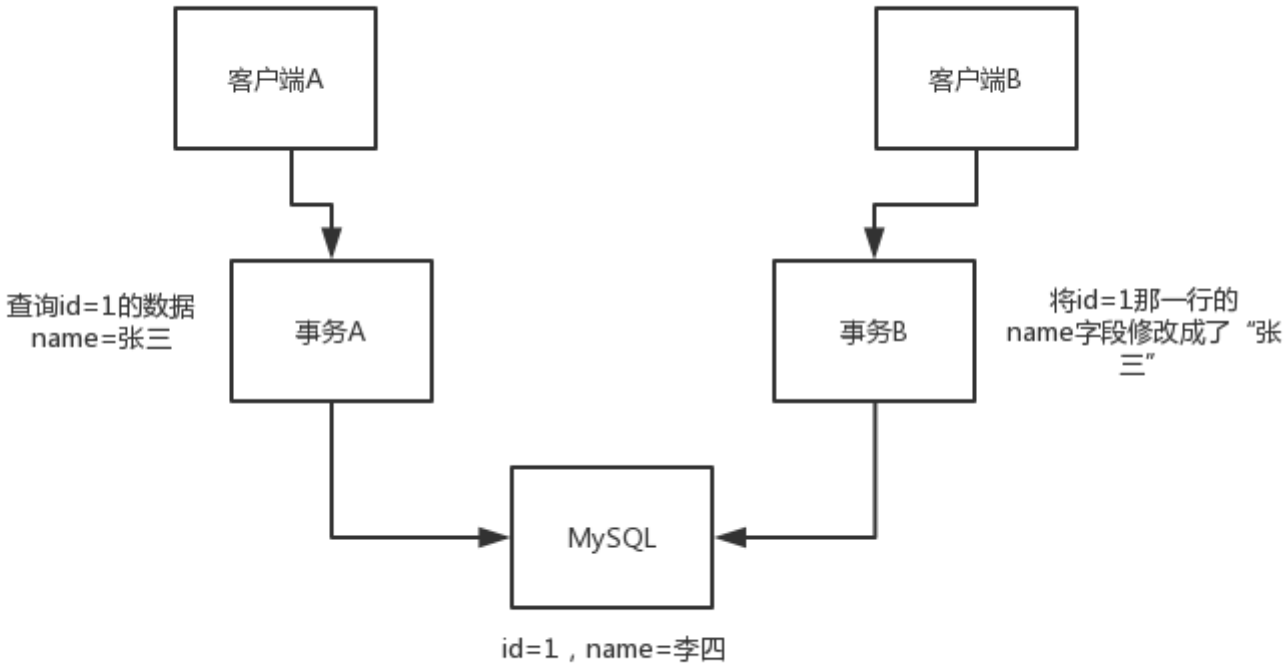
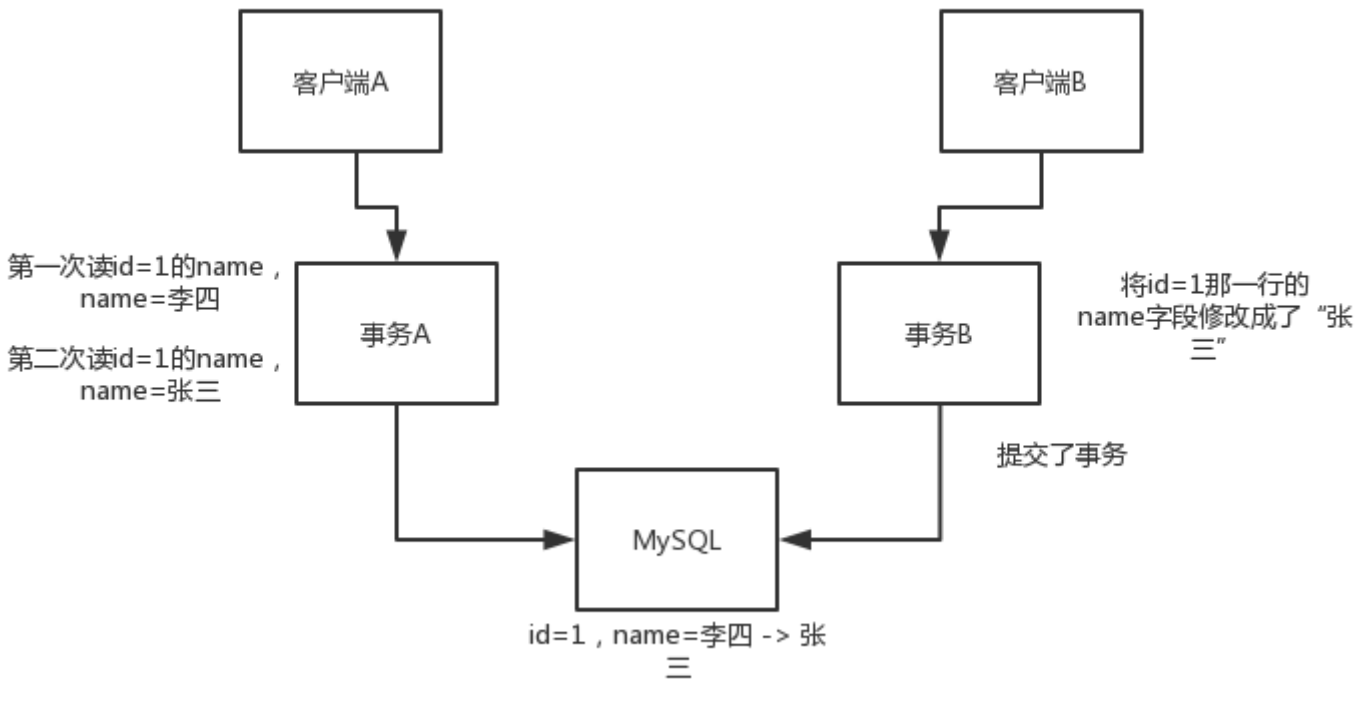
事务隔离级别

- 读未提交



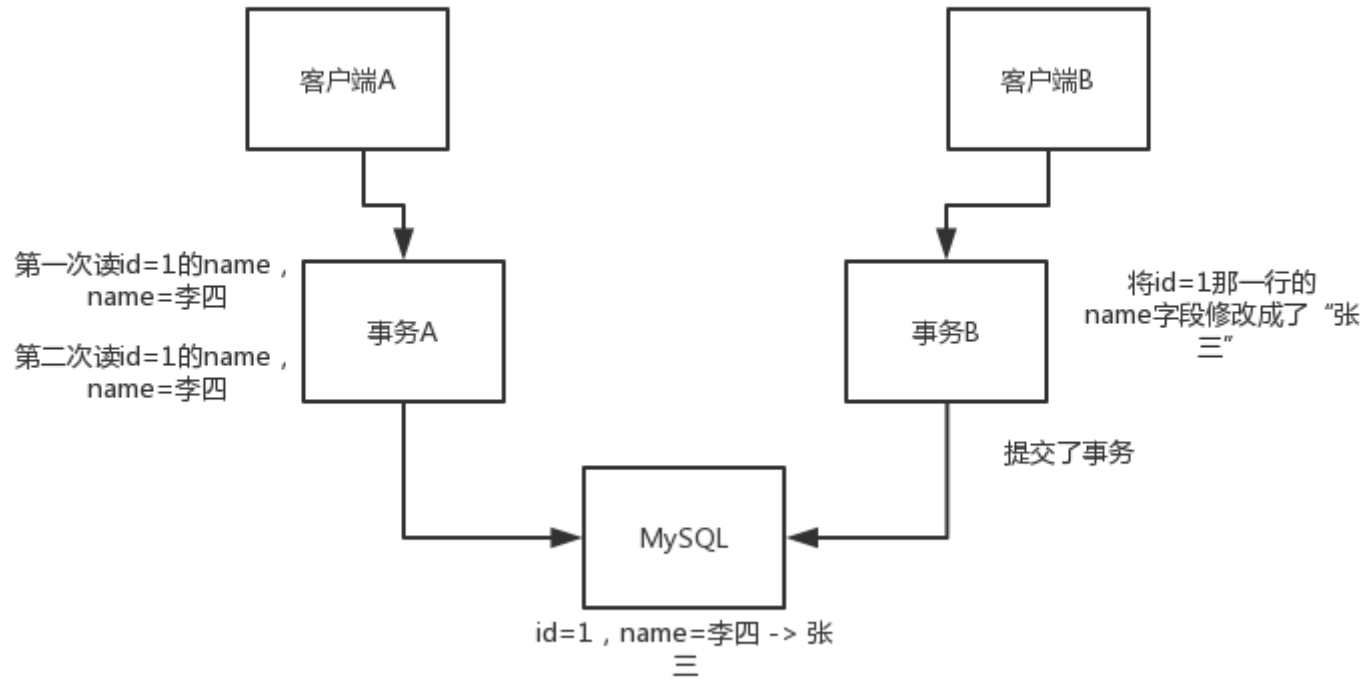
事务B还没提交的时候，修改的数据，就让事务A给读到了，这个也叫做脏读

- 读已提交



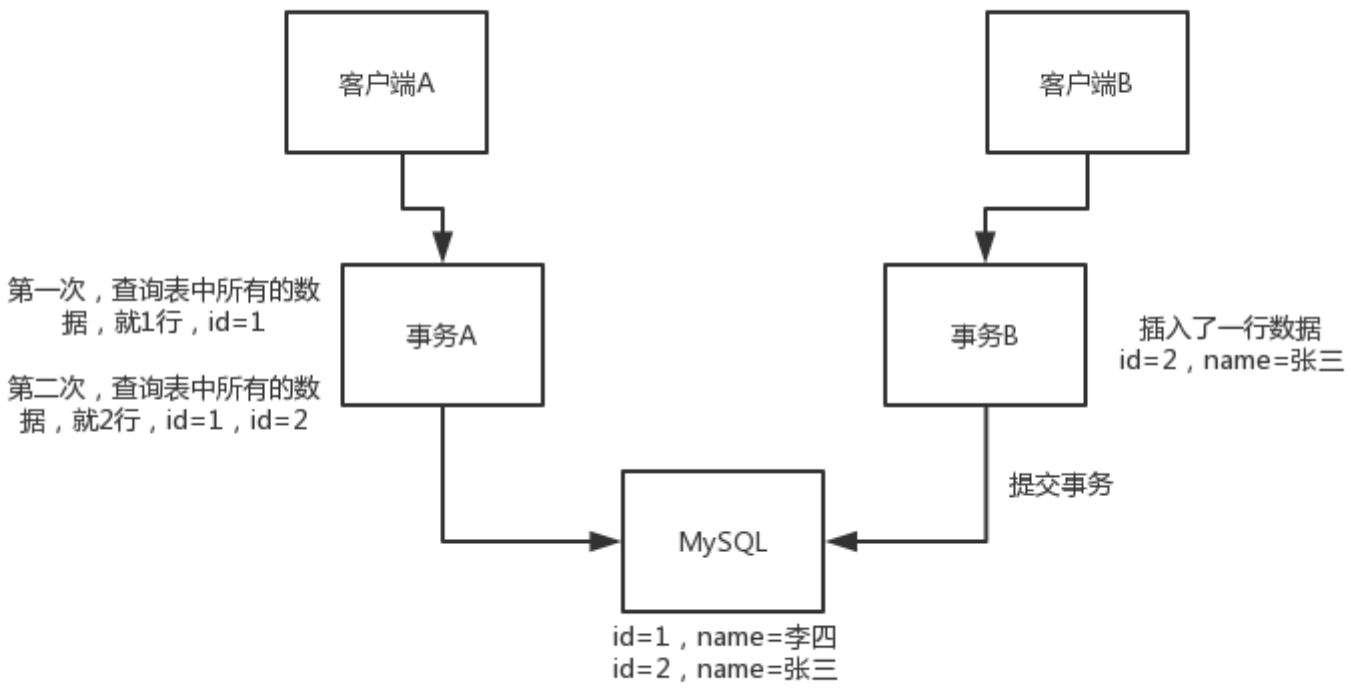
事务A在跑的时候， 先查询了一个数据是值1，然后过了段时间，事务B把那个数据给修改了一下还提交了，此时事务A再次查询这个数据就成了值2了，这是读了人家事务提交的数据啊，所以是读已提交。这个也叫做不可重复读，就是所谓的一个事务内对一个数据两次读，可能会读到不一样的值。

- 可重复读

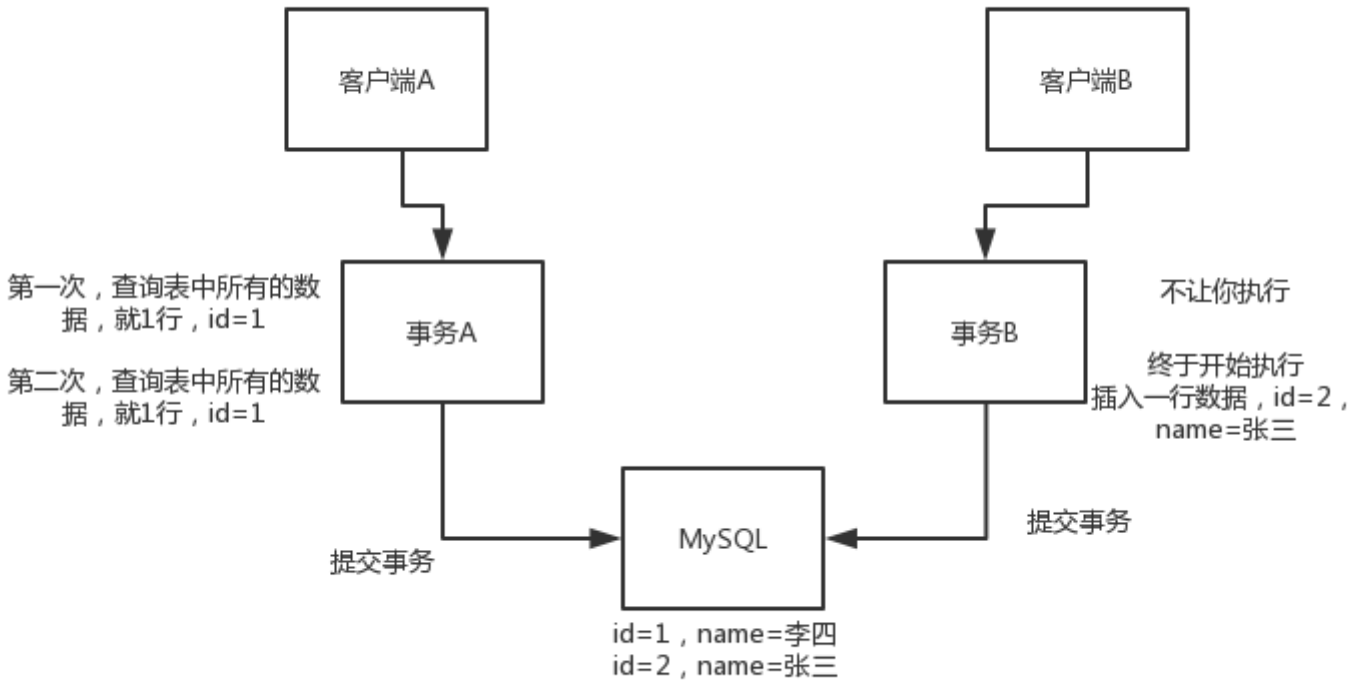


事务A在执行过程中，对某个数据的值，无论读多少次都是值1；哪怕这个过程中事务B修改了数据的值还提交了，但是事务A读到的还是自己事务开始时这个数据的值

- 串行化



（幻读）



（串行化）

- spring事务传播机制

```
public class ServiceA {
    @Autowired
    private ServiceB b;
    @Transactional
```

```
    public void methodA() {
    // 一坨数据库操作
    for(int i = 0; i < 51; i++) {
    try {
        b.methodB();
    } catch(Exception e) {
        // 打印异常日志
    }
}
    // 一坨数据库操作
    }

}

public class ServiceB {
    @Transactional(propagation= PROPAGATION_REQUIRES_NEW)
    public void methodB() throws Exception {
        // 一坨数据库操作
    }
```

（1）PROPAGATION_REQUIRED：这个是最常见的，就是说，如果ServiceA.method调用了ServiceB.method，如果ServiceA.method开启了事务，然后ServiceB.method也声明了事务，那么ServiceB.method不会开启独立事务，而是将自己的操作放在ServiceA.method的事务中来执行，ServiceA和ServiceB任何一个报错都会导致整个事务回滚。这就是默认的行为，其实一般我们都是需要这样子的

（2）PROPAGATION_SUPPORTS：如果ServiceA.method开了事务，那么ServiceB就将自己加入ServiceA中来运行，如果ServiceA.method没有开事务，那么ServiceB自己也不开事务

（3）PROPAGATION_MANDATORY：必须被一个开启了事务的方法来调用自己，否则报错

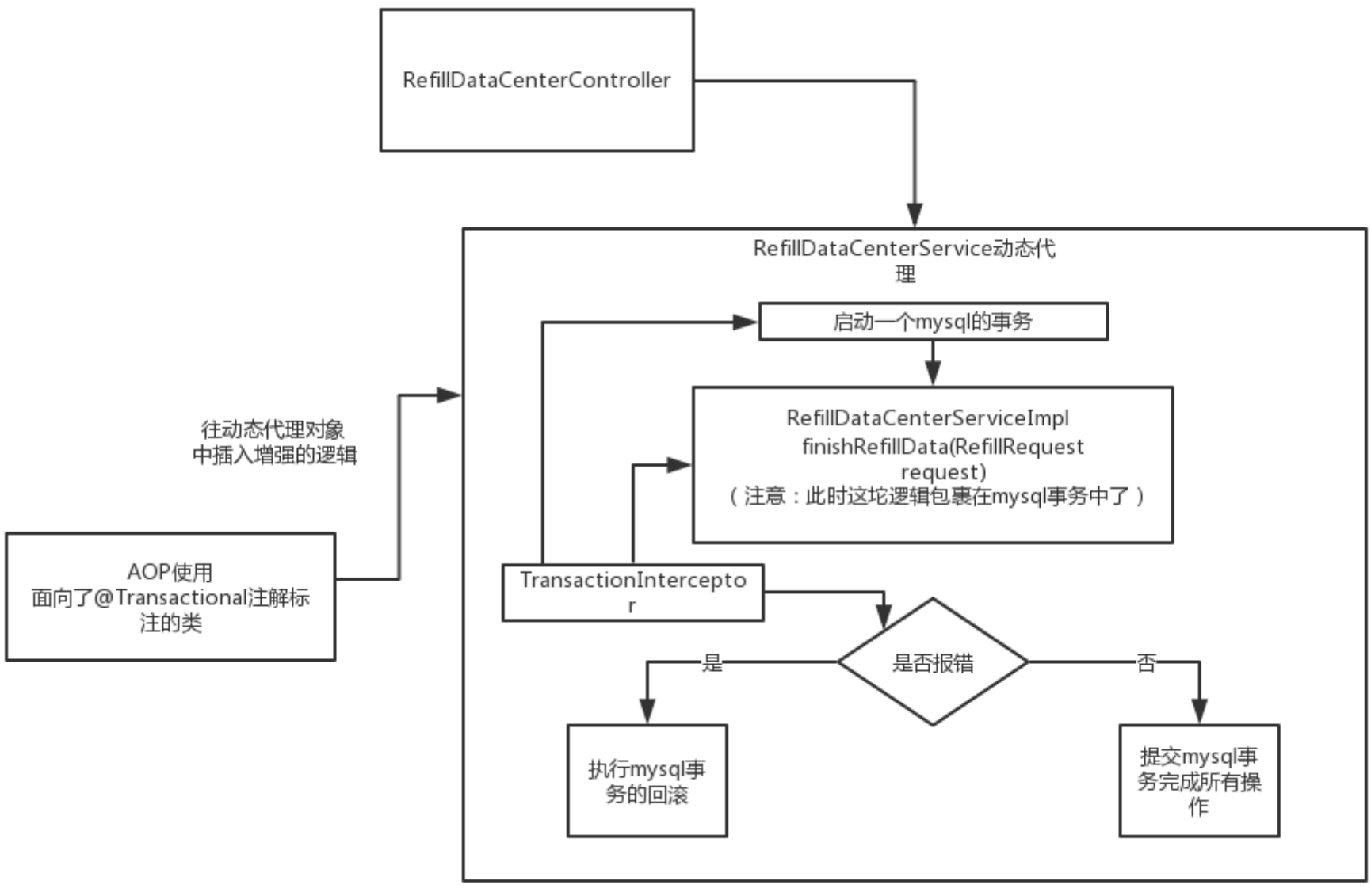
（4）PROPAGATION_REQUIRES_NEW：ServiceB.method强制性自己开启一个新的事务，然后ServiceA.method的事务会卡住，等ServiceB事务完了自己再继续。这就是影响的回滚了，如果ServiceA报错了，ServiceB是不会受到影响的，ServiceB报错了，ServiceA也可以选择性的回滚或者是提交

（5）PROPAGATION_NOT_SUPPORTED：就是ServiceB.method不支持事务，ServiceA的事务执行到ServiceB那儿，就挂起来了，ServiceB用非事务方式运行结束，ServiceA事务再继续运行。这个好处就是ServiceB代码报错不会让ServiceA回滚

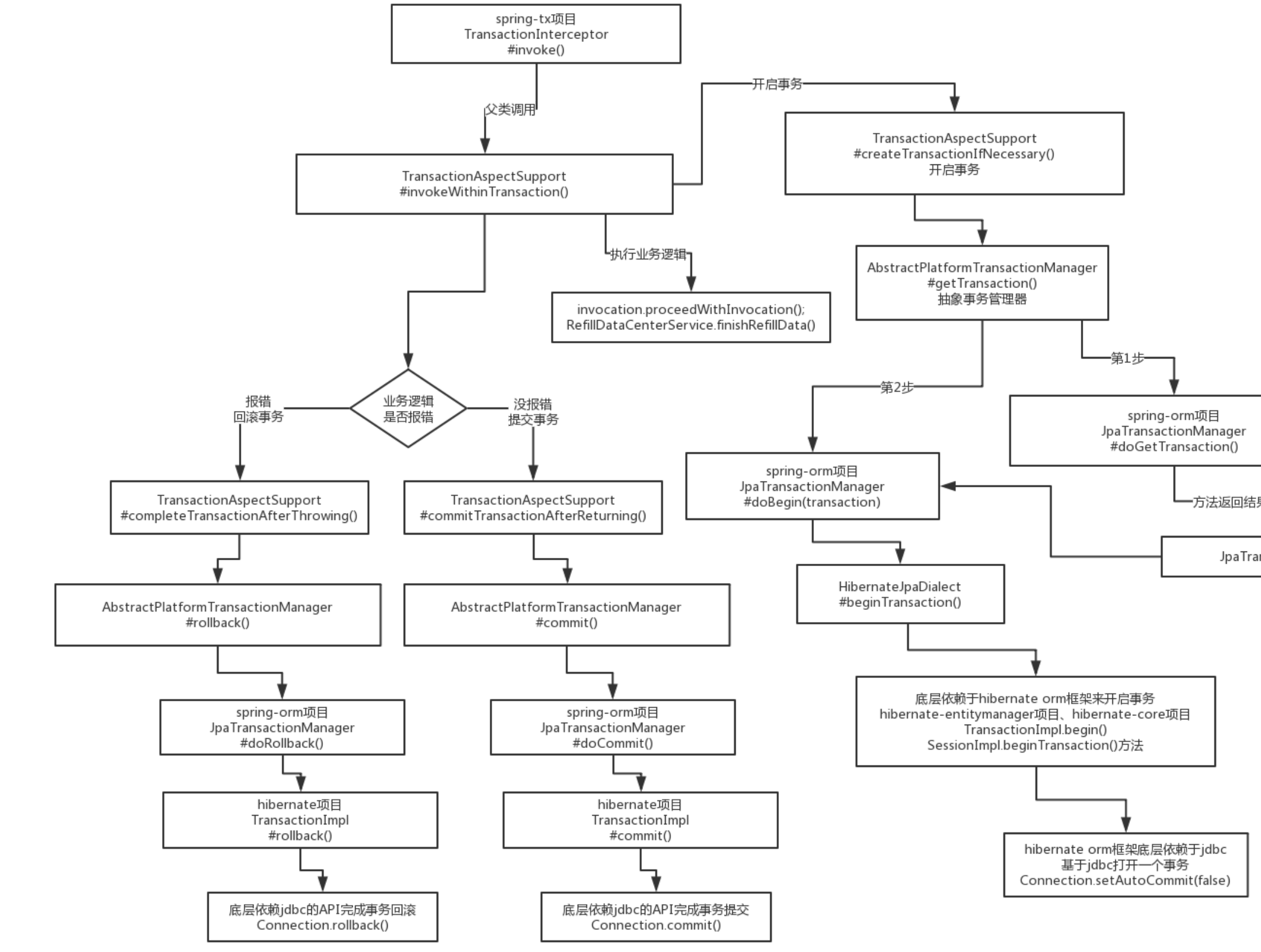
（6）PROPAGATION_NEVER：不能被一个事务来调用，ServiceA.method开事务了，但是调用了ServiceB会报错

（7）PROPAGATION_NESTED：开启嵌套事务，ServiceB开启一个子事务，如果回滚的话，那么ServiceB就回滚到开启子事务的这个save point

- spring事务基本原理

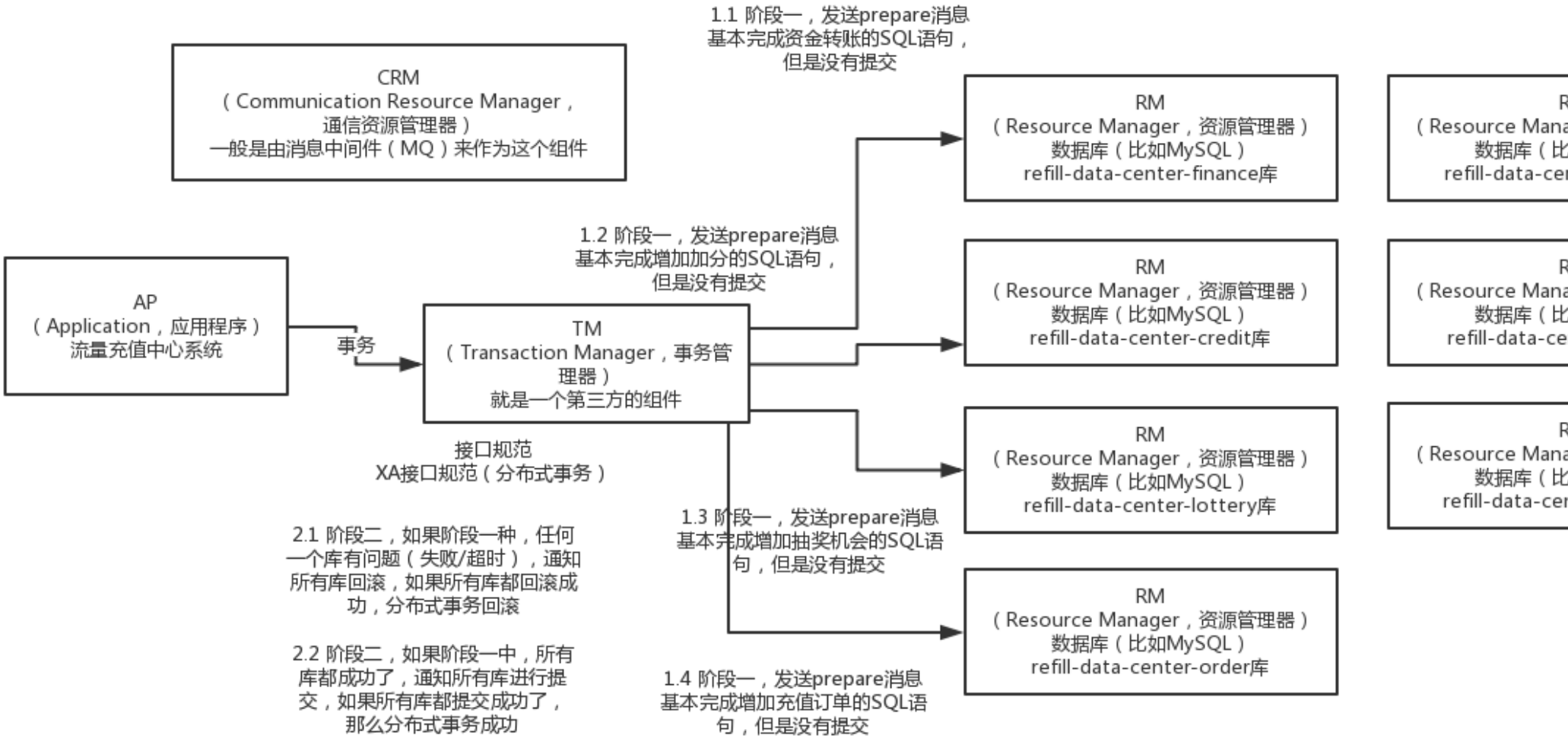


- spring事务源码

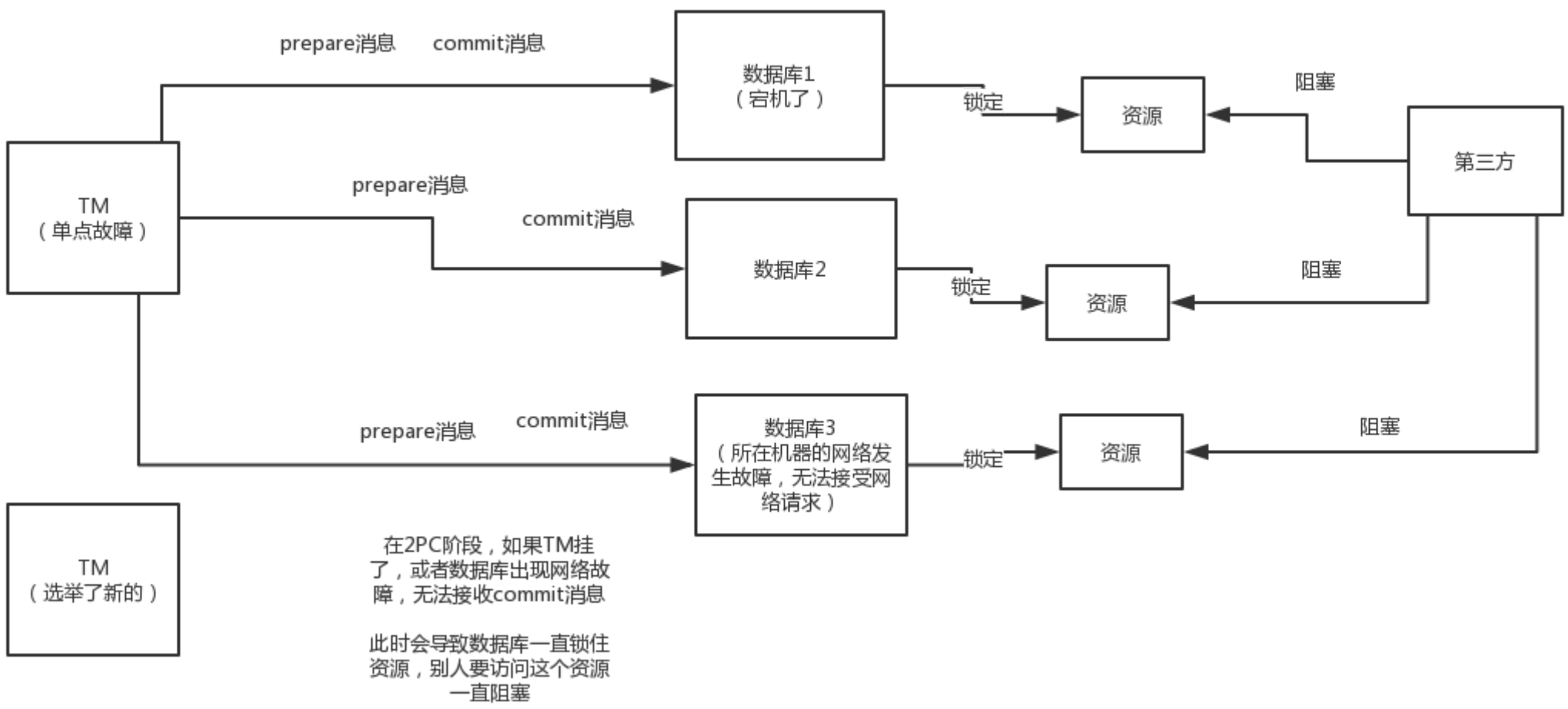


spring的事务的支持是使用的hibernate这个框架中的一些事务管理相关的一些底层的类库和代码，最终依赖于jdbc的API来开启事务，Connection.setAutoCommit(false)，基于jdbc的Connection连接执行了setAutoCommit(false)，将自动提交设置为false，相当于是在底层发送了一条指令给mysql说，开启事务

- XA规范与2PC协议（单系统多库系统）



- 2PC存在的问题



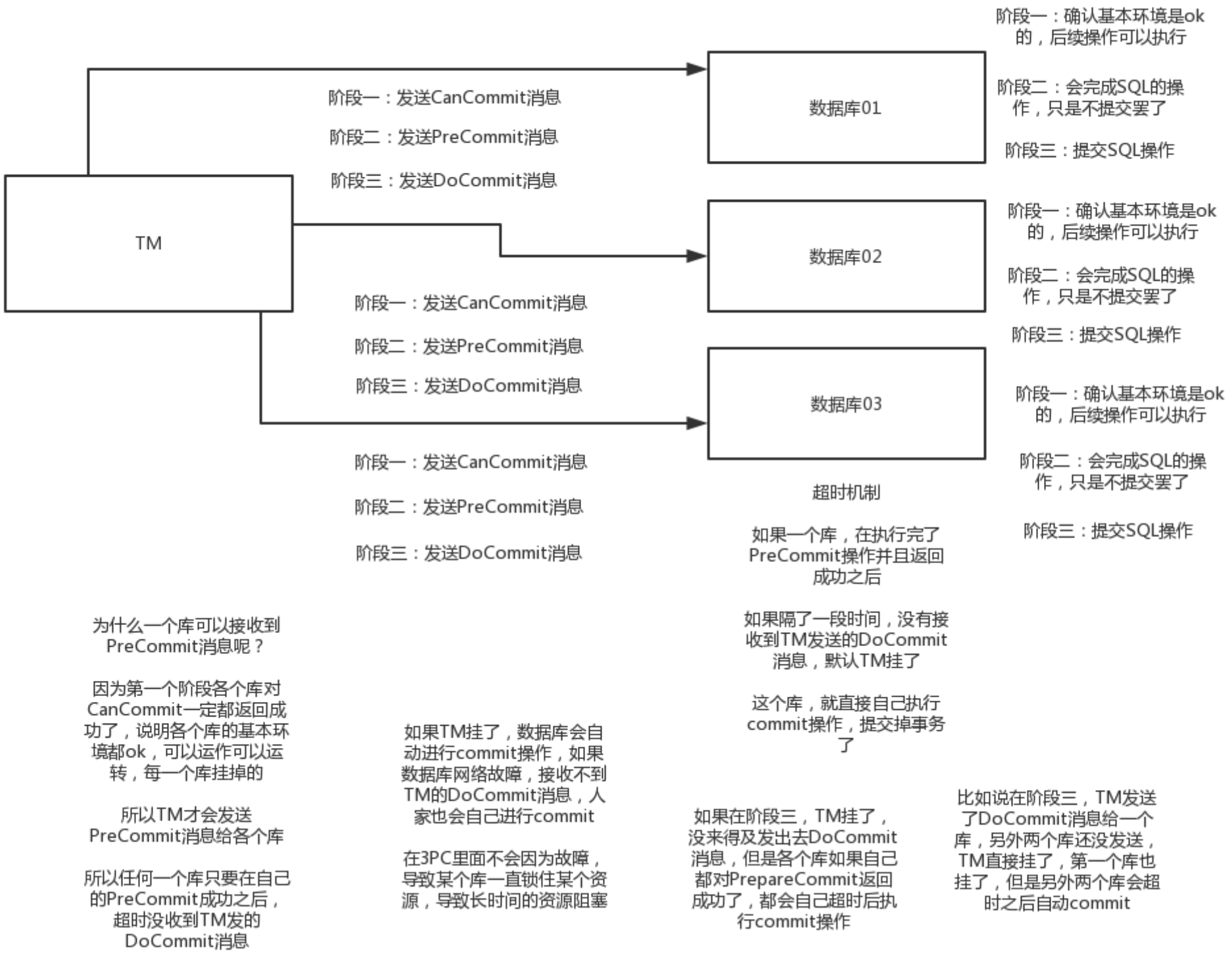
1、同步阻塞：在阶段一里执行prepare操作会占用资源，一直到整个分布式事务完成，才会释放资源，这个过程中，如果有其他人要访问这个资源，就会被阻塞住

2、单点故障：TM是个单点，一旦挂掉就完蛋了

3、事务状态丢失：即使把TM做成一个双机热备的，一个TM挂了自动选举其他的TM出来，但是如果TM挂掉的同时，接收到commit消息的某个库也挂了，此时即使重新选举了其他的TM，压根儿不知道这个分布式事务当前的状态，因为不知道哪个库接收过commit消息，那个接收过commit消息的库也挂了

4、脑裂问题：在阶段二中，如果发生了脑裂问题，那么就会导致某些数据库没有接收到commit消息，那就完蛋了，有些库收到了commit消息，结果有些库没有收到，这咋整呢，那肯定完蛋了

- 3PC



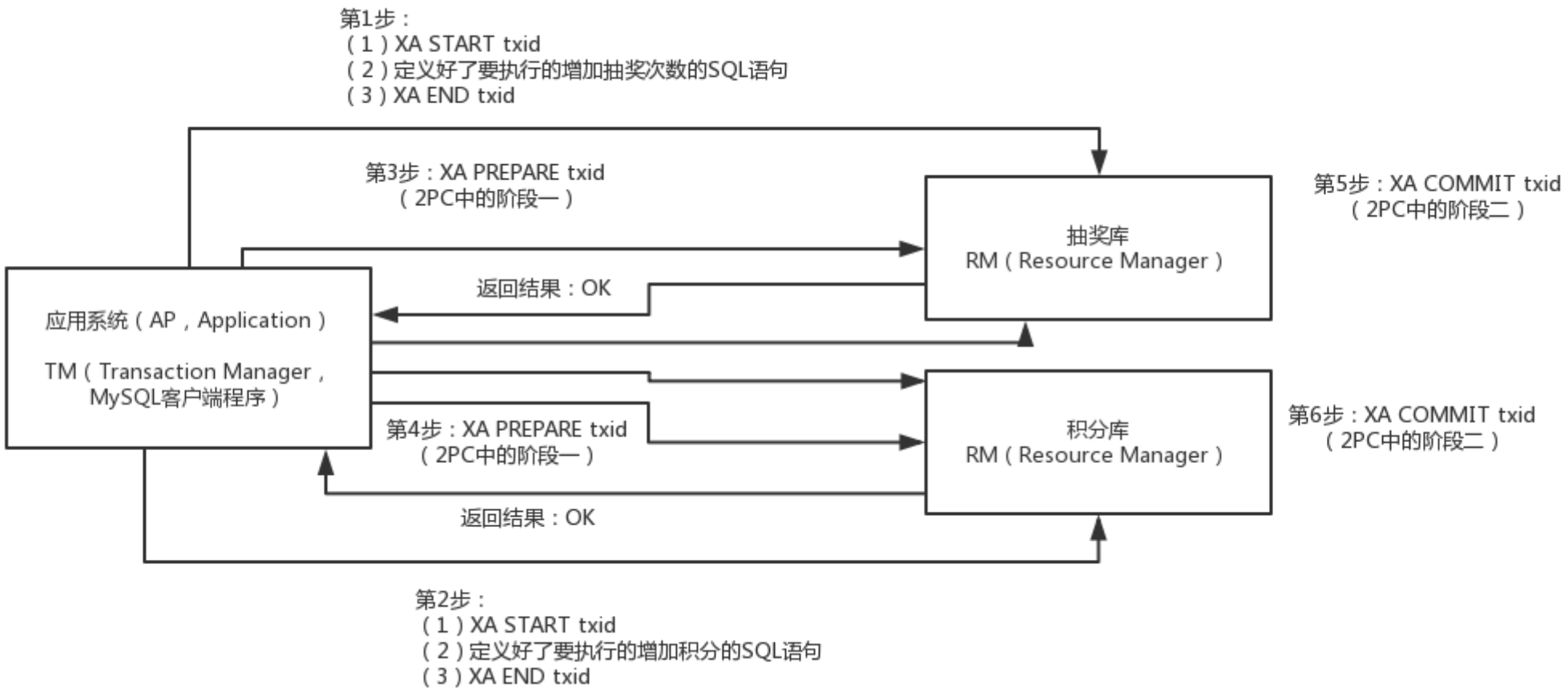
（1）CanCommit阶段：这个就是TM发送一个CanCommit消息给各个数据库，然后各个库返回个结果，注意一下，这里的话呢，是不会执行实际的SQL语句的，其实说白了，就是各个库看看自己网络环境啊，各方面是否ready

（2）PreCommit阶段：如果各个库对CanCommit消息返回的都是成功，那么就进入PreCommit阶段，TM发送PreCommit消息给各个库，这个时候就相当于2PC里的阶段一，其实就会执行各个SQL语句，只是不提交罢了；如果有个库对CanCommit消息返回了失败，那么就尴尬了，TM发送abort消息给各个库，大家别玩儿了，结束这个分布式事务

（3）DoCommit阶段：如果各个库对PreCommit阶段都返回了成功，那么发送DoCommit消息给各个库，就说提交事务吧，兄弟们，各个库如果都返回提交成功给TM，那么分布式事务成功；如果有个库对PreCommit返回的是失败，或者超时一直没返回，那么TM认为分布式事务失败，直接发abort消息给各个库，说兄弟们回滚吧，各个库回滚成功之后通知TM，分布式事务回滚。在DoCommit阶段，各个库自己也有超时机制，也就是说，如果一个库收到了PreCommit自己还返回成功了，等了一会儿，如果超时时间到了，还没收到TM发送的DoCommit消息或者是abort消息，直接判定为TM可能出故障了，人家库自己颠儿颠儿的就执行DoCommit操作，提交事务了，这样就解决了TM单点故障问题。另外资源阻塞问题也能减轻一下，因为一个库如果一直接收不到DoCommit消息，不会一直锁着资源，人家自己会提交释放资源的，所以能减轻资源阻塞问题。

缺陷：如果人家TM在DoCommit阶段发送了abort消息给各个库，结果因为脑裂问题，某个库没接收到abort消息，自己还执行了commit操作，所以没法完全保证分布式事务

- MySQL XA分布式事务的实现



- 分布式事务常见的几种方案

（1）XA分布式事务，一般用于单系统多库的场景，当然要是多系统多库，也可以，但是就很麻烦了，不适用于这个方案了

（2）TCC方案，try-confirm-cancel方案，适合于多个服务的操作都比较快，相当于是一堆同步服务调用的操作，包裹在一个事务里面，TCC方案应对的其实是大量的同步服务调用的复杂的事务场景，如果要用TCC来保证分布式事务的执行，一般来说尽量确保每个服务的调用都比较快，一般来说确保一个TCC分布式事务的执行，大概需要总共1秒以内的时间，特别适用于微服务场景，尤其是dubbo技术栈，spring cloud技术栈，跟这些微服务技术栈整合

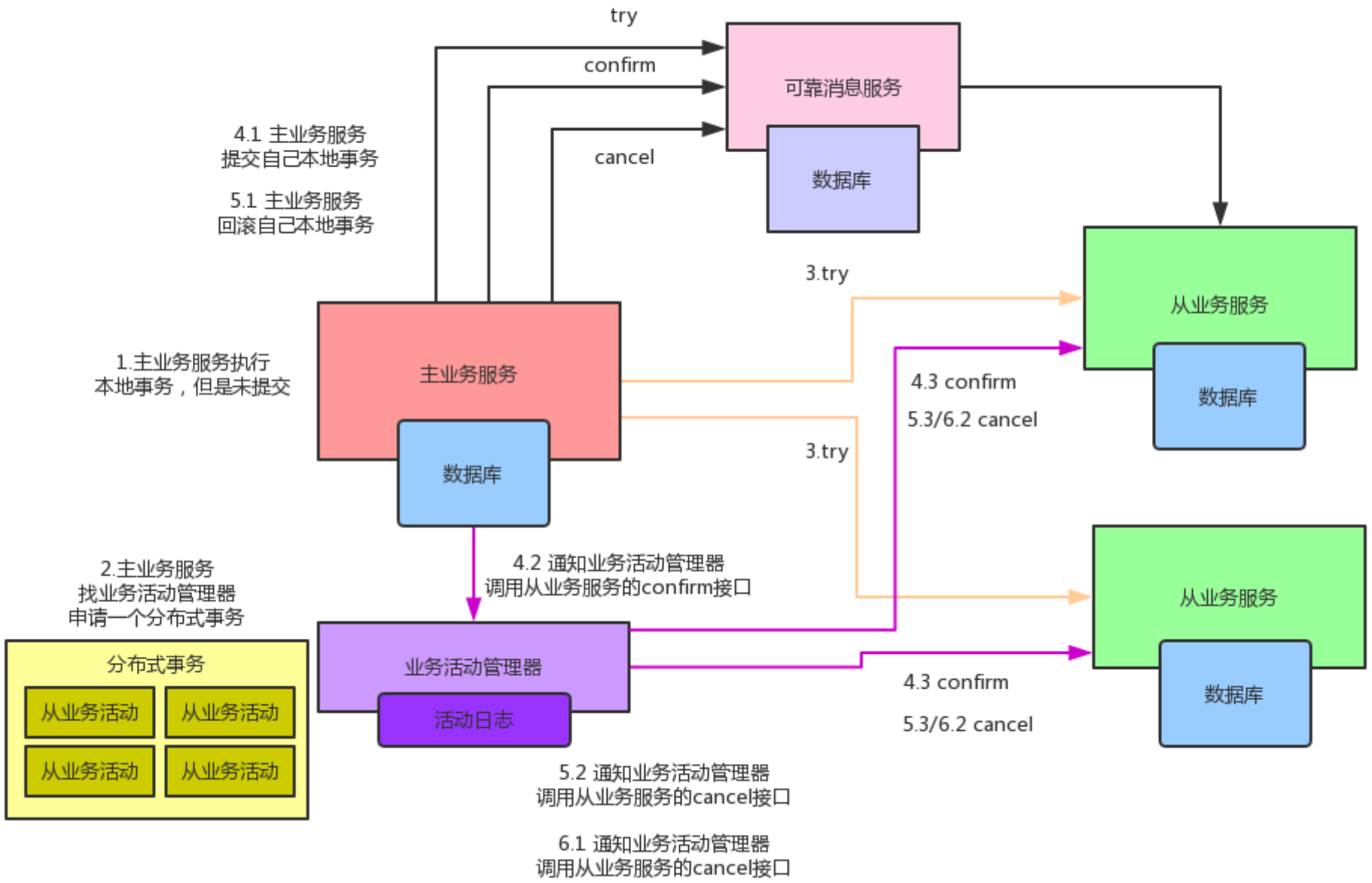
（3）可靠消息最终一致性方案，都不能叫做分布式事务的方案，而是分布式一致性的方案，适合于那那种比较耗时的操作，但一定要成功，通过这个消息中间件做成异步调用，发送一个消息出去，人家服务消费消息来执行业务逻辑，CAP理论，C（最终一致性），也就是说包裹在一个事务中的多个操作，其中有些操作可能在一定时间内是没执行的，可能要等过一段时间之后，然后才能去执行，最终一定会执行的，最终一致性的方案，通过MQ消息中间件保证消息的可靠性，最终来实现最终一致性的方案

（4）最大努力通知方案，不一定保证最终一定会成功，可能会失败，但是他会尽力给你去通知那个服务的执行，比较适合那种不太核心一些服务调用的操作，比如说消息服务

（5）适合长事务（分布式）的sagas方案，基本不用

- TCC方案

1. 通用性TCC技术方案（以流量充值中心为例）



（1）主业务服务：相当于流量充值中心的服务，他就是TCC事务的主控服务，主要控制的服务，负责整个分布式事务的编排和管理，执行，回滚，都是他来控制

（2）从业务服务：相当于我们的资金服务、订单服务、积分服务、抽奖服务、流量券服务，主要就是提供了3个接口，try-confirm-cancel，try接口里是锁定资源，confirm是业务逻辑，cancel是回滚逻辑

（3）业务活动管理器：管理具体的分布式事务的状态，分布式事务中各个服务对应的子事务的状态，包括就是他会负责去触发各个从业务服务的confirm和cancel接口的执行和调用

a. try阶段，资源的锁定，先冻结掉用户的账户资金，将一部分资金转出到冻结资金字段里去；可以创建一个充值订单，但是状态是“交易中”

b. confirm阶段，就是将用户的冻结资金减掉，转移到商户的账户里去；同时将充值订单的状态修改为“交易成功”；完成抽奖机会、积分、流量券的新增

c. cancel阶段，try阶段任何一个服务有问题的话，那么就cancel掉，相当于是将冻结的资金还回去，将订单状态修改为“交易失败”；如果confirm阶段任何一个服务有问题的话，也是cancel掉，相当于是将商户账户里的资金还到用户账户里去，同时将订单的状态修改为“交易失败”

上图流程为：

（1）主业务服务会先在本地开启一个本地事务（这个本地事务说白了，就是你的主业务服务是不是也可能会干点儿什么事儿）

（2）主业务服务向业务活动管理器申请启动一个分布式事务活动，主业务服务向业务活动管理器注册各个从业务活动

（3）接着主业务服务负责调用各个从业务服务的try接口

（4）如果所有从业务服务的try接口都调用成功的话，那么主业务服务就提交本地事务，然后通知业务活动管理器调用各个从业务服务的confirm接口

（5）如果有某个服务的try接口调用失败的话，那么主业务服务回滚本地事务，然后通知业务活动管理器调用各个从业务服务的cancel接口

（6）如果主业务服务触发了confirm操作，但是如果confirm过程中有失败，那么也会让业务活动管理器通知各个从业务服务cancel

（7）最后分布式事务结束

2. 异步确保型TCC技术方案

在主业务服务和从业务服务之间加了一个可靠消息服务，将消息放在数据库里的，主业务服务的try、confirm和canel操作都调用可靠消息服务，然后可靠消息服务在try阶段插入一条消息到本地数据库；接着主业务服务执行confirm操作，可靠消息服务就是根据之前的消息，调用从业务服务实际的业务接口；如果要是这个调用失败的话，那么主业务服务发起cancel，可靠消息服务删除自己本地的消息即可，这种方案最大的优点，就是不需要从业务服务配合改造，提供try、confirm和cancel三个接口了。

3. 补偿性TCC解决方案

跟通用型的TCC方案类似的，只不过从业务服务就提供俩接口就ok了，Do和Compensate，就是执行接口和补偿接口，这种方案的好处就是折中一下了，不需要从业务服务改造出来一个T接口，就是锁定资源的接口，只需要加一个补偿接口，如果业务逻辑执行失败之后，进行补偿，这样就可以少做一个接口了，但是因为没有做资源的一个锁定，那么大家需要自己注意类似资金转账的余额检查之类的事儿了，还有就是补偿的时候，因为你

没做资源锁定，所以要注意一下补偿机制是否一定会成功。Do接口，Compensate接口，不要try接口，不要锁定资源，直接执行业务逻辑，如果有失败就调用Compensate接口，补偿接口，回滚刚才的操作。

4. TCC方案细节

（1）接口拆分

需要提供3个接口，try接口里，一般就是预留资源，如果没有资源锁定的操作，try接口就留空，confirm就是原来的业务方法，cancel接口，要提供回滚的方法，就是把try或者confirm里的操作给他回滚了。

（2）接口的几种特殊情况

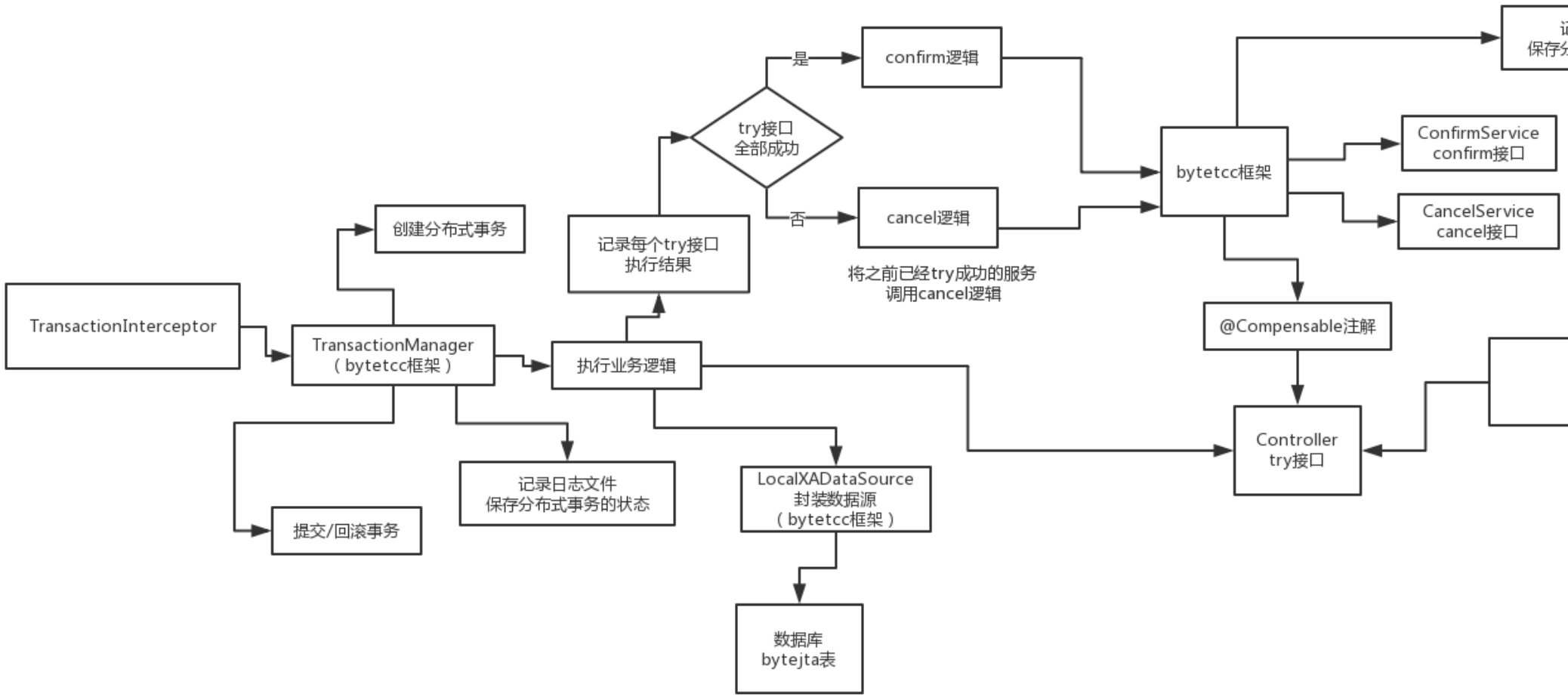
2.1 空回滚：那要是try阶段，比如网络问题，人家压根儿没调通你的try接口，结果就认定失败，直接调用你的cancel接口，咋办？所以你这个时候啥都不能干

2.2 try回滚以及confirm回滚：try阶段如果执行了，但是其他服务try失败了，那么会调用cancel来回滚，你要可以回滚掉try阶段的操作；confirm阶段要是你执行了，但是有别的服务失败了，此时你就要回滚掉confirm阶段的操作

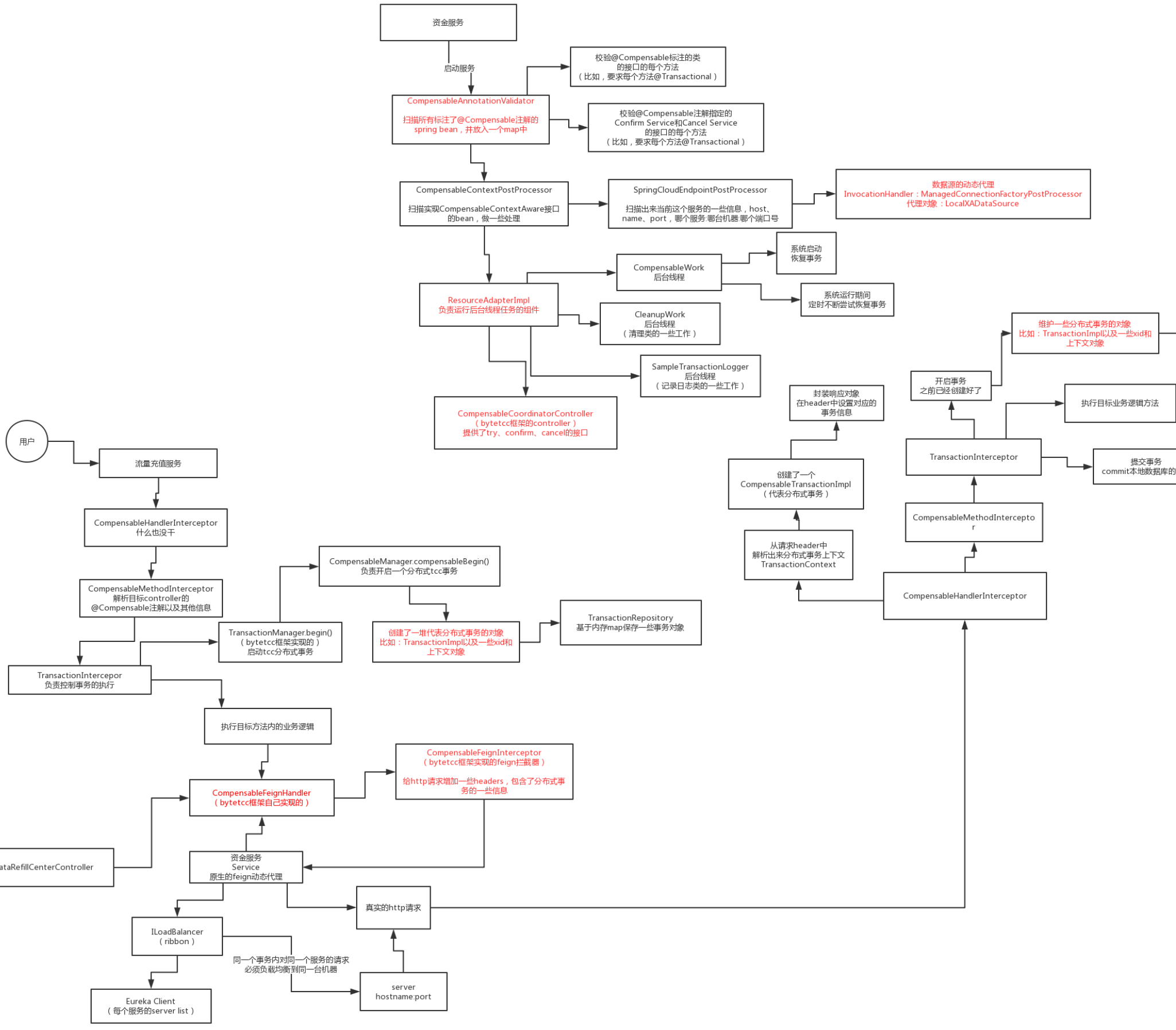
2.3 倒置请求：比如说人家调用try接口，中间网络超时了，结果认定失败，直接调用cancel空回滚了；结果过了几秒钟try接口请求到来，此时咋整呢？尴尬了吧，你要在这个时候不允许执行try接口操作；同理啊，confirm请求超时了，结果都cancel掉了，但是过了几秒请求来了，让你confirm

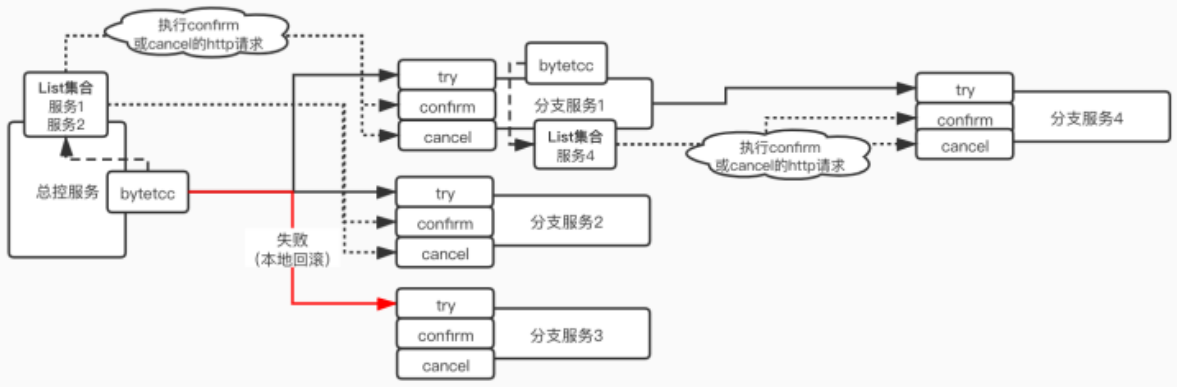
2.4 接口幂等性，try、confirm和cancel都可能被多次调用，所以无论怎么样，你都得保证这几个接口的幂等性

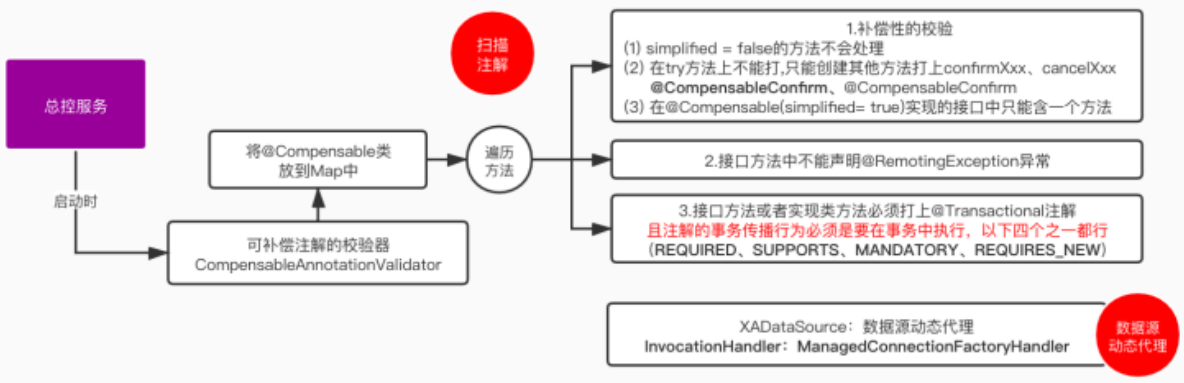
- TCC事务框架实现原理

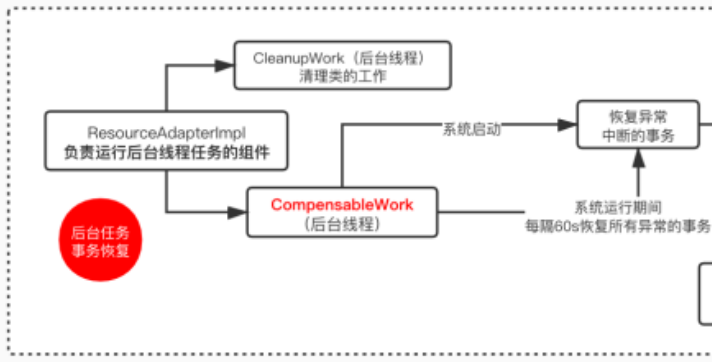


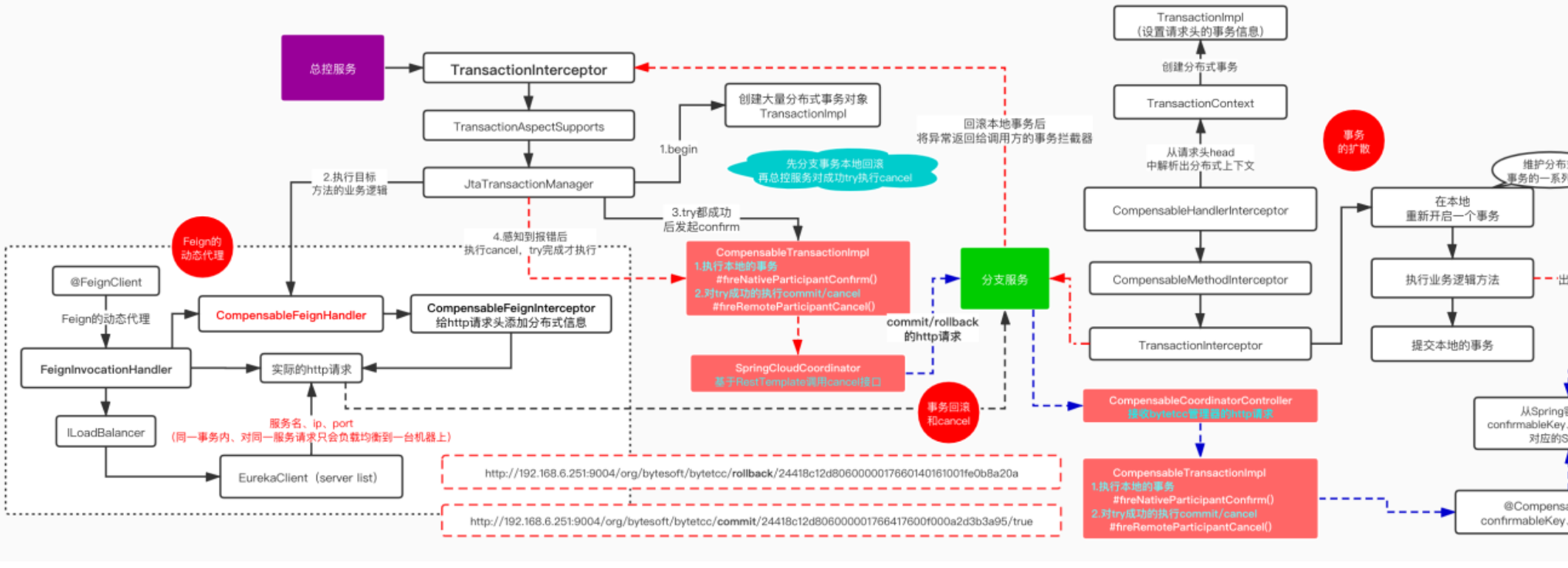
- ByteTcc源码架构


资金服务
启动服务
校验@Compensable标注的类
的接口的每个方法
（比如，要求每个方法@Transactional）
CompensableAnnotationValidator
扫描所有标注了@Compensable注解的
spring bean，并放入一个map中
校验@Compensable注解指定的
Confirm Service和Cancel Service
的接口的每个方法
（比如，要求每个方法@Transactional）
CompensableContextPostProcessor
扫描实现CompensableContextAware接口
的bean，做一些处理
SpringCloudEndpointPostProcessor
扫描出来当前这个服务的一些信息，host、
name、port，哪个服务.哪台机器.哪个端口号
数据源的动态代理
InvocationHandler：ManagedConnectionFactoryPostProcessor
代理对象：LocalXADataSource
CompensableWork
后台线程
系统启动
恢复事务
系统运行期间
定时不断尝试恢复事务
ResourceAdapterImpl
负责运行后台线程任务的组件
CleanupWork
后台线程
（清理类的一些工作）
SampleTransactionLogger
后台线程
（记录日志类的一些工作）
CompensableCoordinatorController
（bytetcc框架的controller）
提供了try、confirm、cancel的接口
用户
流量充值服务
CompensableHandlerInterceptor
什么也没干
CompensableMethodInterceptor
解析目标controller的
@Compensable注解以及其他信息
CompensableManager.compensableBegin()
负责开启一个分布式tcc事务
TransactionManager.begin()
（bytetcc框架实现的）
启动tcc分布式事务
创建了一堆代表分布式事务的对象
比如：TransactionImpl以及一些xid和
上下文对象
TransactionRepository
基于内存map保存一些事务对象
TransactionIntercepor
负责控制事务的执行
执行目标方法内的业务逻辑
CompensableFeignInterceptor
（bytetcc框架实现的feign拦截器）
给http请求增加一些headers，包含了分布式事
务的一些信息
CompensableFeignHandler
（bytetcc框架自己实现的）
DataRefillCenterController
资金服务
Service
原生的feign动态代理
真实的http请求
ILoadBalancer
（ribbon）
同一个事务内对同一个服务的请求
必须负载均衡到同一台机器
server
hostname:port
Eureka Client
（每个服务的server list）
封装响应对象
在header中设置对应的
事务信息
创建了一个
CompensableTransactionImpl
（代表分布式事务）
从请求header中
解析出来分布式事务上下文
TransactionContext
维护一些分布式事务的对象
比如：TransactionImpl以及一些xid和
上下文对象
开启事务
之前已经创建好了
执行目标业务逻辑方法
TransactionInterceptor
提交事务
commit本地数据库的
CompensableMethodIntercepto
r
CompensableHandlerInterceptor

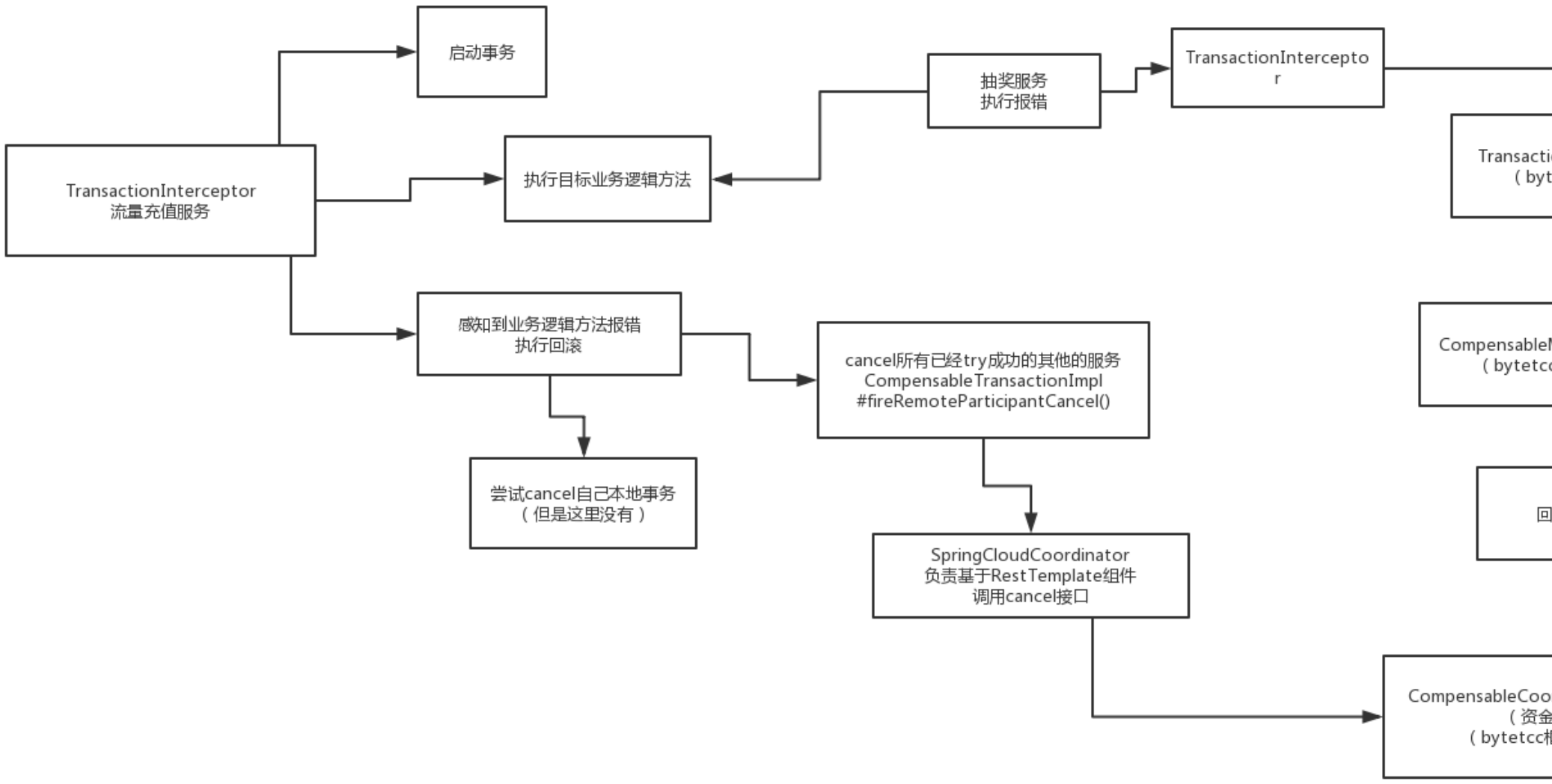




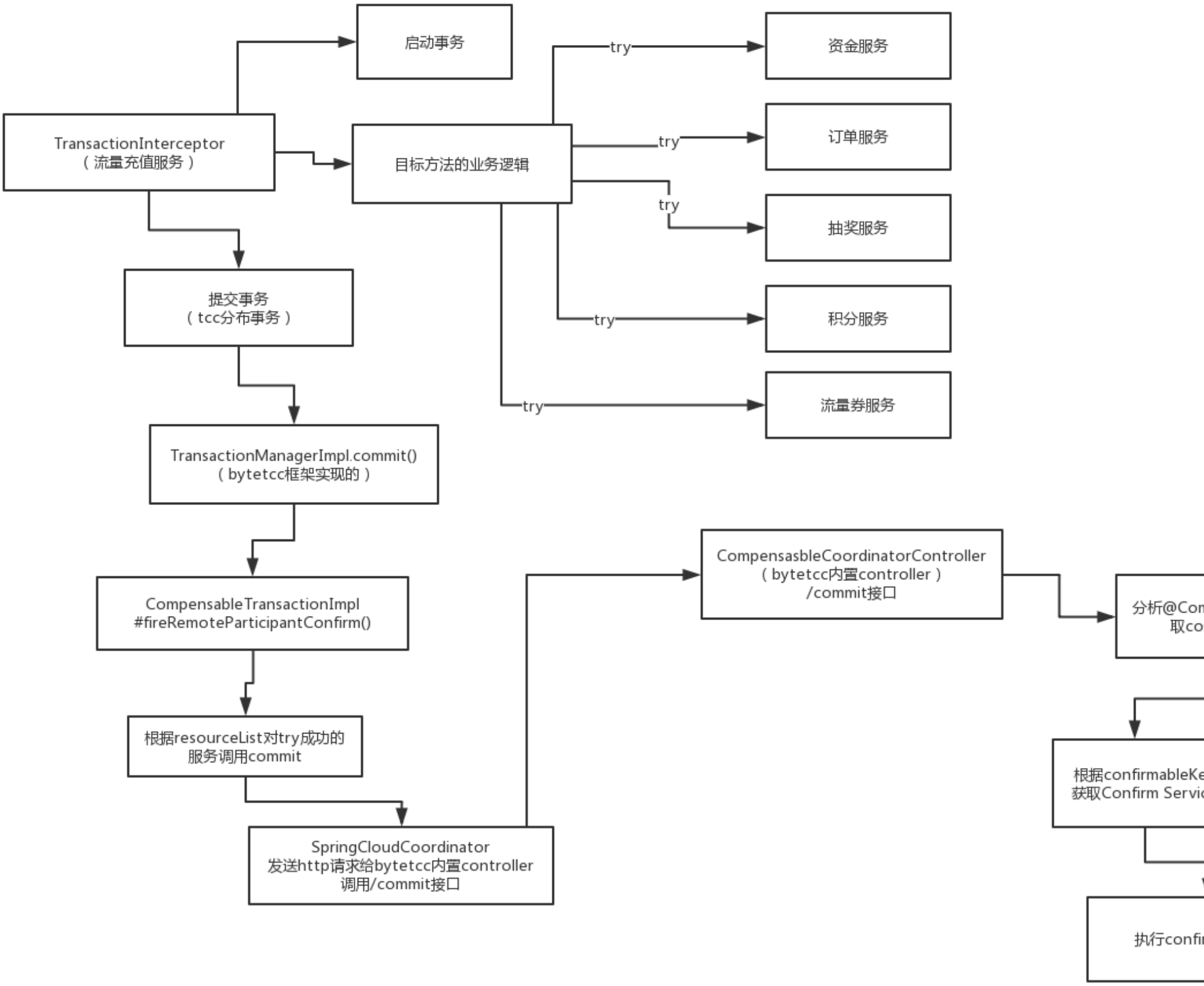




- ByteTcc Cancel源码架构



- ByteTcc Confirm源码架构



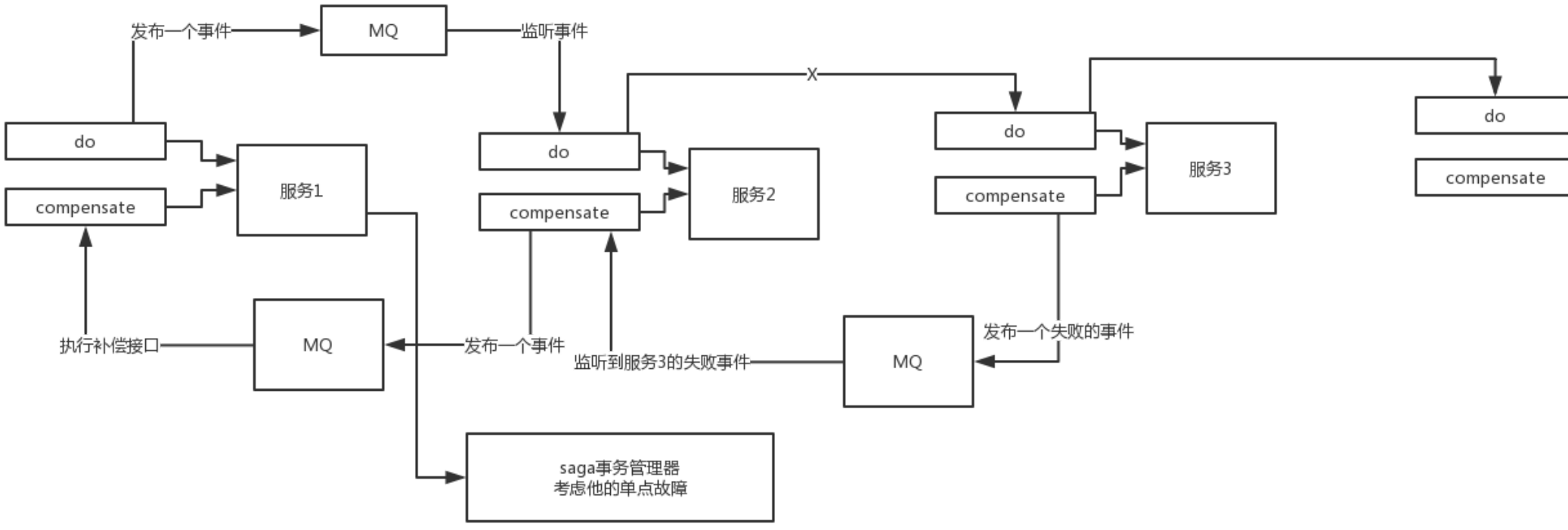
- 如果confirm或cancel执行失败了tcc事务框架是如何不断重试的

通过CompensableWork组件完成，主要有2个作用，一是在系统每次刚启动的时候，对执行到一半儿的事务，还没执行结束的事务，进行恢复，继续执行这个分布式事务，二是分布式事务中所有服务的try都成功了，然后执行confirm，其他服务的confirm都成功了，可能就1个服务的confirm失败了，此时CompensableWork会每隔一段时间，定时不断的去重试那个服务的confirm接口，直到那个服务的confirm接口执行成功，才能认为这个分布式事务执行成功了。confirm或者是cancel有某个服务没有执行成功，bytetcc框架的CompensableWork会不断的去进行重试，CompensableWork会每隔60s去调用一次你的/commit接口或者是/rollback接口，尝试去完成你的分布式事务中的confirm操作或者是cancel操作，CompensableWork每隔60s会去调用一次这个组件的timeingRecovery()方法，这个里面会去调用一个recoverCommit()方法，这个方法底层会去执行SpringCloudCoordinator的方法，去发送一个http请求，调用资金服务的/commit接口

- tcc分布式事务执行到了一半服务挂了重启时如何恢复事务

在重启的过程中，对没完成的分布式事务进行恢复的一个过程，也是通过timingRecover()方法实现的，在刚开始启动的时候，这个方法就会执行一次，就承担了系统重启恢复事务的过程，读取自己的分布式事务的活动日志，找出来没完成的分布式事务，对里面没完成的服务，发起对应的confirm/cancel接口的调用就ok了，如果还是没法成功，后续timeingRecover会不断的去执行对应的接口的调用，来完成这个事务

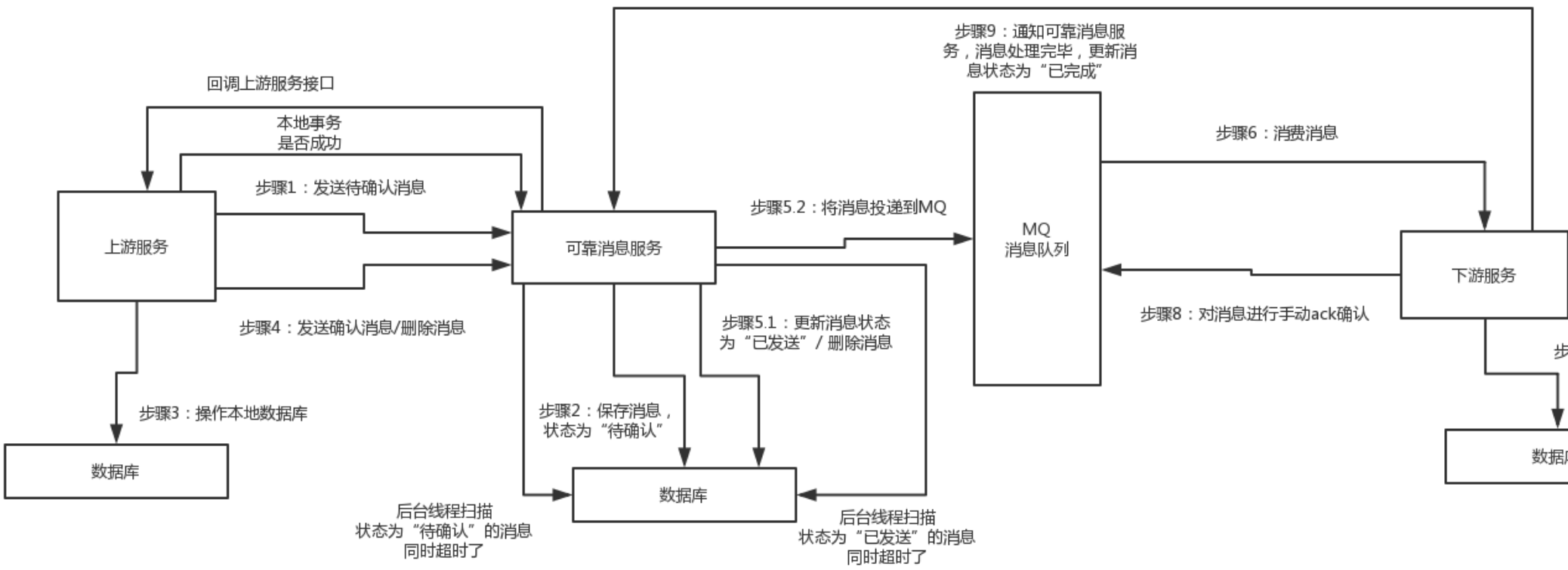
- Saga事务原理



saga是将每个接口，拆分为2个接口，一个是业务接口，另外一个是补偿接口，相当于就是说将tcc里面的try和confirm合并为一个接口，就是先执行业务接口，直接就尝试完成整个业务逻辑的操作，然后如果在服务调用链条里面，某个服务的业务接口执行失败了，那么直接对已经执行成功的所有服务都调用其补偿接口，将之前执行成功的业务逻辑给回滚。saga事务的思想，就是将一个长的分布式事务，拆分为一连串的每个服务的本地事务，然后每个服务对每个接口提供两个接口，一个是业务接口，一个是回滚的补偿接口，正常情况下就是依次的进行调用。它的思想是消息驱动的，是异步的。

优缺点：saga主要是两种思想，一种是编排模式，一种是命令模式。编排模式，优点在于去中心化；缺点在于太麻烦了，如果你有连续几十个服务连续调用，会导致服务对消息的监听非常复杂，而且不好调试和定位问题，服务调用异常，服务调用链的问题排查特别麻烦，引入一个MQ，特别不好，你的服务执行依赖于MQ完成各个服务的调用，引入了复杂性。第二种就是命令模式，就是某个服务如果要开启一个saga分布式事务，那么就得自己搞一个类似saga流程管理器的东西，然后来负责依次调用和执行各个服务，如果某个服务调用失败，就对之前调用成功的服务依次执行补偿接口，saga分布式事务管理器，来管理整个saga事务的调用和执行的流程，如果某个节点失败了，那么就会回滚，重新执行补偿接口，优点在于系统比较简单，定位和处理问题都比较简单，不涉及复杂的消息监听什么的，缺点的话，主saga事务流程管理器单点故障问题

- 可靠消息最终一致性方案



具体的执行流程如下：

（1）上游服务发送一个待确认消息给可靠消息服务

（2）可靠消息服务将这个待确认的消息保存到自己本地数据库里，保存起来，但是不发给MQ，这个时候消息的状态是“待确认”

（3）上游服务操作本地数据库

（4）上游服务根据自己操作本地数据库的结果，来通知可靠消息服务，可以确认发送消息了，或者是删除消息。操作完本地数据库之后，会有两个结果，第一个结果是操作失败了，第二个结果是操作成功了，如果本地数据库操作失败了，本地操作会回滚，回滚之后，上游服务就要通知可靠消息服务删除消息；如果本地数据库操作成功了，那么此时本地事务就提交了，接着就可以通知可靠消息服务发送消息

（5）可靠消息服务将这个消息的状态修改为“已发送”，并且将消息发送到MQ中间件里去，这个环节是必须包裹在一个事务里的，如果发送MQ失败报错，那么可靠消息服务更新本地数据库里的消息状态为“已发送”的操作也必须回滚，反之如果本地数据库里的消息状态为“已发送”，那么必须成功投递消息到MQ里去

```
@Transactional
public void confirmMessage(Long messageId){
messageDAO.updateStatus(messageId,MessageStatus.SENT);
```

```
rabbitmqProducer.send(message);
}
```

如果更新数据库里的消息状态报错了，那么消息根本不会投递到MQ里去；如果更新数据库里的消息状态成功了，但是事务还没提交，然后将消息投递到MQ里去报错了，此时事务管理器会感知到这个异常，然后会直接回滚掉整个事务，更新数据库里消息状态的操作也会回滚掉的，就可以保证说，更新数据库里的消息状态和投递消息到MQ，要么一起成功，要么一起失败。MQ，rabbitmq，都有事务消息的一个实现，你可以先去投递一个prepare的消息，接着如果说数据库操作成功过了，那么就commit那个消息发送给rabbitmq，然后如果数据库操作失败了，就通知mq去rollback一条消息，但是MQ的事务消息最好别轻易用，因为那个性能实在是太低了，吞吐量太差。

（6）下游服务从MQ里监听到一条消息

（7）下游服务根据消息，在自己本地操作数据库

（8）下游服务对本地数据库操作完成之后，对MQ进行ack操作，确认这个消息处理成功

（9）下游服务对MQ进行ack之后，再给可靠消息服务发送个请求，通知该服务说，ok，我这里处理完毕了，可靠消息服务收到通知之后，将消息的状态修改为“已完成”

异常情况处理：

如果1和2两个步骤失败了，应该怎么处理呢？这个是你的上游服务自己应该去做的一个重试的机制，如果你发现调用服务失败或者人家给你返回一个处理失败的结果，你可以做几次重试，一般重试几次就可以了，3这个步骤，如果成功了皆大欢喜，如果失败了，他也会调用可靠消息服务的接口通知一下，让人家删除消息。

如果4和5两个步骤失败了，会呈现出的一个场景就是说，消息在可靠消息服务的数据库里的状态是“待确认”，一直是待确认，从来不会改变了，该如何处理：

（1）可靠消息服务得开一个后台线程，专门扫描那些数据库里处于“待确认”状态的消息，同时该消息的创建时间到现在已经超过了比如10分钟了，这个10分钟是你自己设定的一个超时阈值，一般来说，用个几分钟，或者10分钟都行

（2）发现那种一直处于“待确认”状态的消息，还超过了一定的时间，就认为是超时了

（3）对超时的消息，需要回调上游服务特意提供的查询这个操作状态的一个接口，然后上游服务自己判断一下这个操作是否执行，举个例子，比如说上游服务本来应该是将订单状态修改为“交易成功”的，此时回调过去以后，上游服务就得自己去判断下那个操作执行了没有，查下那个订单的状态

（4）如果操作是还没执行，那么证明是3那个步骤失败了，就是上游服务的本地数据库操作失败了，所以导致消息一直处于“待确认”的状态，此时可靠消息服务需要将这条消息给删除即可

（5）如果操作是已经执行了，那么说明是4或者5失败了，要不是上游服务没通知到可靠消息服务，要不是可靠消息服务自己没成功投递出去消息，此时可靠消息服务就是再次尝试用一个事务来更新本地消息状态为“已发送”，同时尝试再次发送消息给MQ

如果6789出现了问题该如何处理：

（1）可以开一个后台线程，专门监控“已发送”的消息，如果超过了10分钟，那么就需要判定为超时了

（2）这个时候也是需要可靠消息服务再次重新投递消息给到MQ，让下游服务再次去消费

（3）下游服务的接口一定要保证幂等性，数据库操作只能成功的执行一次

- 最大努力通知方案